Compiler Bug Isolation via Effective Witness Test Program Generation[*]

Junjie Chen[†]
College of Intelligence and Computing, Tianjin University, China
junjiechen9208@gmail.com

Jiaqi Han
Peiyi Sun
HCST (Peking University), MoE, China
{hanjiaqi,spy}@pku.edu.cn

Lingming Zhang
University of Texas at Dallas, USA
zhanglm10@gmail.com

Dan Hao[‡]
Lu Zhang
HCST (Peking University), MoE, China
{haodan,zhanglucs}@pku.edu.cn

ABSTRACT

Compiler bugs are extremely harmful, but are notoriously difficult to debug because compiler bugs usually produce few debugging information. Given a bug-triggering test program for a compiler, hundreds of compiler files are usually involved during compilation, and thus are suspect buggy files. Although there are lots of automated bug isolation techniques, they are not applicable to compilers due to the scalability or effectiveness problem. To solve this problem, in this paper, we transform the compiler bug isolation problem into a search problem, i.e., searching for a set of effective witness test programs that are able to eliminate innocent compiler files from suspects. Based on this intuition, we propose an automated compiler bug isolation technique, DiWi, which (1) proposes a heuristic-based search strategy to generate such a set of effective witness test programs via applying our designed witnessing mutation rules to the given failing test program, and (2) compares their coverage to isolate bugs following the practice of spectrum-based bug isolation. The experimental results on 90 real bugs from popular GCC and LLVM compilers show that DiWi effectively isolates 66.67%/78.89% bugs within Top-10/Top-20 compiler files, significantly outperforming state-of-the-art bug isolation techniques.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Compiler Debugging, Bug Isolation, Test Program Generation

ACM Reference Format:
Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler Bug Isolation via Effective Witness Test Program Generation. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338957

1 INTRODUCTION

Software bugs in modern software systems can incur huge cost. For example, `Tricentis.com` studied software bugs in 363 companies all over the world, and reported that these bugs incurred almost $1.1 Trillion cost, and affected over 4.4 Billion customers in 2016. Among software bugs, compiler bugs are especially critical, since almost all software systems are compiled via compilers and a buggy compiler can potentially affect all the software systems built on it. Therefore, it is crucial to detect, isolate, and fix compiler bugs.

Although researchers have devoted dedicated efforts to compiler testing [15, 24, 29, 43, 53, 58, 82], compiler bug isolation and fixing are still a tedious and time-consuming process, since compilers are very complex and developers have to understand the root cause of a compiler bug and then determine the fixing strategy. In the literature, lots of *automated* bug isolation techniques (also known as fault localization, e.g., spectrum-based techniques or SBFL [22, 38, 52, 72, 77], slicing-based techniques [81], mutation-based techniques [35, 51, 60, 66, 67, 87]) have been proposed. However, these techniques can hardly be applicable to compilers due to their scalability or effectiveness problem. First, compilers like GCC are complex and large, making it extremely expensive to perform advanced static/dynamic analysis on a compiler. Therefore, program-analysis based techniques like slicing-based techniques and mutation-based techniques can hardly be used in compiler bug isolation. For example, we found that it can take over a month to execute only a limited set of GCC compiler mutants. Second, due to the complexity of compilers, the execution traces between passing and failing test programs tend to differ significantly. Therefore, it is hard for SBFL, which isolates compiler bugs by comparing coverage between passing and failing test programs, to isolate compiler bugs effectively, which is also demonstrated in Sections 2 and 5. In other words, automated compiler bug isolation is still challenging due to its inherent difficulty.

Given a bug-triggering test program (also called failing test program) for a compiler, developers need to manually isolate buggy compiler files (i.e., source files of compilers) among all the touched compiler files when compiling the given failing test program. However, there are usually at least hundreds of compiler files involved in

[*] This work is partially supported by the National Key Research and Development Program of China under Grant No. 2017YFB1001803, the National Natural Science Foundation of China under Grant Nos. 61872008, 61672047, and 61861130363; it is also partially supported by NSF grants CCF-1566589, CCF-1763906, and Amazon.

[†] This work was done when he was in Peking University.

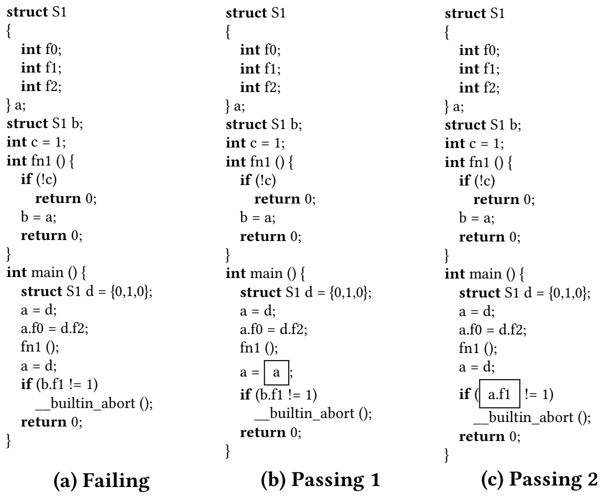
[‡] Corresponding author. HCST is short for Key Lab of High Confidence Software Technologies.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338957

the compilation. All these files are suspects. SBFL has demonstrated that passing tests can be regarded as witnesses to help reduce the suspicion of innocent program elements [7]. In this paper, we call passing test programs that are able to eliminate innocent compiler files from suspects *witness test programs*. However, as demonstrated in Sections 2 and 5, the widely-used developer-provided passing test programs are not effective witnesses, since they can hardly isolate compiler bugs effectively. Therefore, in this paper, we aim to find a set of effective witness test programs. With the set of effective witness test programs and the given failing test program, developers can automatically isolate buggy compiler files precisely. With this intention, we transform the problem of automated compiler bug isolation to a search problem, i.e., finding a set of effective witness test programs to help isolate compiler bugs precisely.

However, it is challenging to find such a set of effective witness test programs. On the one hand, each witness test program in the set is required to have a large witness capability, i.e, it can eliminate innocent compiler files as many as possible from suspects. On the other hand, the witness capabilities of different witness test programs in the set are required to be diverse, i.e., each of them can eliminate different innocent compiler files from suspects, so that grouping them can be helpful to isolate the buggy compiler files precisely. To solve this problem, in this paper, we propose the first compiler bug isolation technique via searching for diversified witnesses, called *DiWi* (**Di**versified **Wi**tnesses). The main contribution of DiWi lies in how to find the set of effective witness test programs. To address the first challenge, we design a series of mutation rules for DiWi to generate witness test programs via slightly mutating the given failing test program. The key insight is that such minor changes are more likely to make the generated witness test program share a close compiler execution trace with the failing test program, so that the generated witness test program can eliminate more innocent compiler files from suspects. To address the second challenge, DiWi utilizes some heuristics to guide the construction of the set of effective witness test programs. More specifically, during the generation of each witness test program, DiWi considers the diversity of compiler execution traces between it and the already generated witness test programs, so that these generated witness test programs can eliminate different innocent compiler files from suspects. Finally, DiWi ranks the files executed by the failing test program based on coverage comparison between the set of effective witness test programs and the given failing test program like SBFL [74].

To evaluate the effectiveness of DiWi, we constructed an extensive dataset including 90 real-world reproducible bugs from the popular GCC [2] and LLVM [5] compilers. The experimental results on all the 90 studied bugs show that DiWi effectively isolates 10/37/60/71 bugs (out of 90) within Top-1/5/10/20 buggy files, significantly outperforming traditional SBFL. As the core of DiWi, we investigated the contribution of our search-based witness test programs compared with the developer-provided test suite shipping with the buggy compiler (which includes test programs passing on this compiler) and the randomly generated passing test programs via Csmith [82] (the most widely-used random C program generation tool). The results show that our search-based witness test programs significantly outperform the latter two, e.g., isolating 150.00% and 66.67% more bugs than the developer-provided passing test programs and randomly generated passing test programs at the Top-1 position. We also investigated the contribution of our heuristic-based search strategy compared with the random search strategy during generation, and explored the synthesis of DiWi and developer-provided test suites. In summary, this paper makes the following contributions:



```
struct S1
{
   int f0;
   int f1;
   int f2;
} a;
struct S1 b;
int c = 1;
int fn1 () {
   if (!c)
      return 0;
   b = a;
   return 0;
}
int main () {
   struct S1 d = {0,1,0};
   a = d;
   a.f0 = d.f2;
   fn1 ();
   a = d;
   if (b.f1 != 1)
      __builtin_abort ();
   return 0;
}
```

(a) Failing

```
struct S1
{
   int f0;
   int f1;
   int f2;
} a;
struct S1 b;
int c = 1;
int fn1 () {
   if (!c)
      return 0;
   b = a;
   return 0;
}
int main () {
   struct S1 d = {0,1,0};
   a = d;
   a.f0 = d.f2;
   fn1 ();
   a = a;
   if (b.f1 != 1)
      __builtin_abort ();
   return 0;
}
```

(b) Passing 1

```
struct S1
{
   int f0;
   int f1;
   int f2;
} a;
struct S1 b;
int c = 1;
int fn1 () {
   if (!c)
      return 0;
   b = a;
   return 0;
}
int main () {
   struct S1 d = {0,1,0};
   a = d;
   a.f0 = d.f2;
   fn1 ();
   a = d;
   if ( a.f1 != 1)
      __builtin_abort ();
   return 0;
}
```

(c) Passing 2

Figure 1: LLVM Bug 24482

- **Idea.** An automated compiler bug isolation technique that transforms the problem of bug isolation to the problem of guided test program generation via search&mutation.
- **Implementation.** A practical open-source tool implementing the proposed compiler bug isolation technique based on the LLVM Clang infrastructure.
- **Benchmark.** An open-source dataset containing 90 reproducible real bugs from GCC and LLVM compilers for future research on compiler bug detection, isolation, and fixing.
- **Study.** An extensive study on all 90 bugs from our dataset demonstrating that the proposed technique is able to significantly outperform existing techniques; our search-based witness test programs (the core of DiWi) have higher quality than the developer-provided test suite and the randomly generated passing test programs for compiler bug isolation; lastly, components from developer-provided test suites can further boost DiWi effectiveness.

2 MOTIVATION

Here, we use an example to illustrate the motivation of this paper. Figure 1 shows an example from LLVM bug ID 24482, where the left is the reported bug-triggering test program. When the buggy compiler (revision 245195 in LLVM trunk) compiles the failing program, it produces different outputs under the compilation options "`-O1`" and "`-Os`". The bug occurs at the file "`DeadStoreElimination.cpp`" due to dead store elimination across basic blocks.

It is tedious for developers to manually find the buggy file since this revision contains a large number of files (i.e., 3,581). Although traditional SBFL is reported to be effective in other software systems [68], it may not be effective in compilers due to their characteristics. We applied Ochiai [7] (one of the most effective formulae in SBFL) based on the failing test program and the developer-provided test suite, and found the buggy file is ranked at the 659_{th} position out of 3,581 files. That is, with traditional SBFL developers need to examine 658 innocent compiler files before finding the buggy one, indicating the inferior effectiveness of traditional SBFL in compilers.

To isolate compiler bugs effectively following SBFL, instead of the developer-provided passing test programs, a set of effective witness test programs are desirable. Intuitively, witness test programs sharing similar execution traces (except the buggy file) with the given failing test program are more helpful to eliminate innocent files from suspects. With this intuition, we generated 20 witness test programs by randomly introducing various minor changes into the failing test program, making them share similar execution trace with the failing one. Then we compared their coverage to isolate the bug like SBFL [74]. In this way, the buggy file is ranked at the 5_{th} position, demonstrating that these generated witness test programs by slightly changing the failing test program can help isolate the bug effectively.

We further analyzed the isolation result using each witness test program and the given failing test program (namely a pair of test programs) to learn the performance of each individual witness test program. Among the 20 pairs, 1 pair achieves the optimal result (i.e., Top-1), 13 pairs rank the buggy file within the first 10 positions, and 7 pairs rank the buggy file after the 30_{th} positions. Overall, there are many effective pairs and a few low-quality pairs, by changing the failing test program minorly. For example, Figure 1b shows the optimal witness test program that ranks the buggy file at the 1_{st} position, which is generated by changing one variable of the failing test program in Figure 1a; while Figure 1c shows a low-quality witness test program that ranks the buggy file at the 33_{rd} position, which is generated by replacing "`b.f1`" with "`a.f1`". Since we use coverage comparison between test programs to isolate compiler bugs, a witness test program eliminating more innocent files from suspects tends to mean that it shares a more similar compiler execution trace with the failing test program, and also means that it has higher quality.

Based on the above analysis, we have the following observations: First, a set of high-quality witness test programs can help isolate compiler bugs with the given failing test program; Second, even if we change the failing test program minorly, the obtained witness test programs are of various quality.

3 RELATED WORK

We discuss the most closely related work to compiler bug isolation.
Automated Debugging. In the literature, there are a huge amount of work on automated debugging, e.g., fault localization [27, 45–47, 55, 56, 80, 85] and program repair [21, 28, 48, 59, 62, 65], where our work targets the former. As presented in Section 1, the existing fault localization techniques cannot work well on compilers.

In the literature there also exists mutation-based fault localization [50, 51, 60, 66, 87], which aims to mutate *source programs* to check the impact of each code element on the test outcomes. However, our work is to introduce the idea of mutation to slightly change *test programs* so as to generate a set of effective witness test programs. Besides, some work focuses on improving fault localization via test generation [8, 10, 11, 54, 71] or test selection [25, 26, 61, 70] for ordinary programs. However, compiler inputs are programs and compilers are extremely complex and huge, making none of these existing test-generation and test-selection techniques for better debugging directly applicable here.

In automated debugging, our work is mostly related to *compiler debugging*. Most compiler debugging work focused on providing debugging messages/visualization [13, 31, 41, 64, 73]. Some work focused on reducing bug-triggering tests to facilitate debugging [12, 32, 69, 76, 84]. In our work, the provided bug-triggering tests are already the reduced ones, as required by compiler developers. Besides, Chen et al. [19] proposed a technique to rank test programs triggering bugs such that test programs triggering distinct bugs are early in the list. Holmes and Groce [34] further proposed a new metric to determine similarity of failing test programs to facilitate debugging. In contrast, our work aims to *automatically isolate compiler bugs* via effective witness test program generation. In particular, Zeller [83] proposed to produce an entire cause-effect chain in GCC from input to result for facilitating compiler debugging. Actually, cause-effect chain and our technique are complementary: 1) the former produces fault-diagnosis information at the program-state level while the latter does this at the source-code level, 2) the former manipulates in memory and may not handle external states, 3) the latter is more lightweight.
Mutation Testing. Mutation testing is one of the most effective methods to measure test-suite quality [36, 39, 86]. It deliberately seeds bugs into the original source program to simulate the bugs that developers often make in practice. Different from it, our work aims to conduct *test program mutation* to generate a set of effective witness test programs for facilitating compiler bug isolation. Here, we both apply existing mutation rules from mutation testing and also design new mutation rules for compiler bug isolation.
Compiler Testing. Compiler testing usually happens before compiler bug isolation (the target of our work). Most research on compiler testing focuses on test program generation [20, 82], test oracle construction [43, 57], and test execution acceleration [14–16, 18]. The general idea of mutation is also applied to compiler testing [20, 33, 42, 44], which aims to generate different test programs as much as possible by *faster diverging from the given seed program* for *detecting deep compiler bugs*. Different from them, our work aims to utilize mutation for *compiler bug isolation*, and the design goal of our mutation is to flip the compiler execution results (i.e., from `failing` to `passing`) to generate a set of witness test programs *close to a given failing test program*.

To sum up, the existing fault localization techniques cannot work well for large-scale compiler systems, while the existing applications of mutation cannot be directly used to solve the problem of compiler bug isolation. Therefore, this work makes the first attempt to isolate compiler bugs via search-based mutation.

4 APPROACH

Following the observations in Section 2, we propose a novel technique, named **DiWi** (**Di**versified **Wi**tnesses), to isolate compiler bugs. In DiWi, we first deliberately generate a set of effective witness test programs, and then compare them with the given failing test program like SBFL to isolate compiler bugs [74]. From Section 2, we know that it is hard to ensure each generated witness test program to be effective, and thus we use an aggregation mechanism to minimize the impact of low-quality witness test programs in DiWi. Therefore, there are two key issues to the success of DiWi. First,

$$
\begin{array}{rl}
a ::= & x \mid n \mid op_u\ a \mid a_1\ op_a\ a_2 \\
b ::= & \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid \\
 & b_1\ op_l\ b_2 \mid a_1\ op_r\ a_2 \\
S ::= & x := a \mid S_1; S_2 \mid \\
 & \textbf{while}\ (b)\ \textbf{do}\ S \mid \\
 & \textbf{if}\ (b)\ \textbf{then}\ S_1\ \textbf{else}\ S_2
\end{array}
$$

Figure 2: Syntax rules for the WHILE language

we need to carefully construct each witness test program to avoid introducing many low-quality witness test programs; otherwise, we have too much noise. Second, the witness test programs should evenly eliminate buggy suspects; otherwise, the aggregation process is prone to biases. To help construct such a set of effective witness test programs, we define the following two criteria.

C_1: Each test program in the set of effective witness test programs should share a similar compiler execution trace with the failing test program.

C_2: Test programs in the set of effective witness test programs should have great diversity in their compiler execution traces.

However, since the space of witness test programs is extremely huge (in fact infinite to be precise), efficiently generating such a set of effective witness test programs satisfying the two criteria is challenging. To satisfy the first criterion, we design a series of mutation rules for DiWi to generate witness test programs by slightly mutating the failing test program. Intuitively, such minor changes are likely to make the generated witness test program share a close compiler execution trace with the failing test program. Here, we call our mutation *witnessing mutation*, aiming to generate witness test programs with a large witness capability. To satisfy the second criterion, DiWi utilizes some heuristics to guide the construction of the set of effective witness test programs. That is, during the generation of a new witness test program, DiWi considers the diversity of compiler execution traces between the new one and the already generated witness test programs, aiming to make their witness capabilities *diversified*.

In the following, we introduce the designed witnessing mutation in Section 4.1, heuristic-based witness test program construction in Section 4.2, and the aggregation mechanism for compiler bug isolation in Section 4.3.

4.1 Witnessing Mutation

As presented in Section 3, the existing applications of mutation cannot be directly used for compiler bug isolation. In our context, the goal of *test program mutation* is to introduce slightly different control- and data-flow information to flip the compiler execution results (i.e., from `failing` to `passing`) to generate witness test programs. Therefore, we design a series of mutation rules specific to our mutation goal.

Mutation Rules. To achieve the mutation goal of flipping the failing test program, we manually analyzed historical compiler bugs to investigate why these test programs are able to trigger bugs and which characteristics of test programs can sensitively impact compiler execution results. Also, to make the changes minor, we design our mutation, based on the learned knowledge, at the level of fine-grained program elements, including *variables*, *operators*, and *constants*. Therefore, we propose three mutation categories: *variable mutation*, *operator mutation*, and *constant mutation*. For *variable mutation*, each program variable can potentially be changed into another compatible variable or type, since program variables are the core of data dependencies and variable types can impact many compiler optimizations. For *operator mutation*, each program operator (i.e., arithmetic, logical, relational, and unary) can potentially be changed into another compatible operator, since program operators can significantly impact data- and control-dependencies. For *constant mutation*, each program constant can potentially be changed into another value, since many compiler bugs are triggered under some specific values. Here, we both apply existing mutation rules from traditional mutation testing [39] and design new mutation rules for compiler bug isolation (e.g., variable type mutation).

More formally, following the presentation of prior work [89] we view a test program as a syntactic skeletal structure $\mathbb{P}$ with "holes" that can be configured with various variables (denoted as hole $\square_v$), operators (denoted as hole $\square_o$), and constant values (denoted as hole $\square_c$). In this way, we can fill each hole with mutated content to derive a new mutated program. For the ease of presentation, let us consider a WHILE-style language that has been widely used in the program analysis research [63, 89]. The program syntax rules for the WHILE-style language are shown in Figure 2. In the rules, non-terminals a, b, and S denote arithmetic expressions, boolean expressions, and program statements, respectively; terminals op_a, op_l, op_r, and op_u denote arithmetic, logical, relational, and unary operators, respectively; terminals x and n denote program variables and constants. Note that we use the WHILE-style language for the ease of presentation, and our technique applies to a full-fledged language such as C. To obtain program mutations, we recursively apply a mutation transformation to the WHILE syntax rules. Figure 3 presents the syntax rules for all our three mutation categories. We denote all the three categories of transformations and program holes as $\llbracket\rrbracket$ and $\square$, respectively, i.e., $\llbracket\rrbracket$=$\llbracket\rrbracket_v \cup \llbracket\rrbracket_o \cup\ \llbracket\rrbracket_c$ and $\square$=$\square_v \cup \square_o \cup \square_c$. In this way, we formally define our mutation process.

Definition 1 (Mutation Skeleton). *Given any test program P, we say $\mathbb{P}$ is a* mutation skeleton *of P iff the abstract syntax tree (denoted as $T_{\mathbb{P}}$) of $\mathbb{P}$ is the same as the transformed abstract syntax tree (denoted as $\llbracket T_P \rrbracket$) of P, i.e., $T_{\mathbb{P}} = \llbracket T_P \rrbracket$.*

Definition 2 (First-order Mutation). *Given a test program P and its mutation skeleton $\mathbb{P}$, for all the n holes (e.g., $\{\square_1, \square_2, ..., \square_n\}$) within $\mathbb{P}$, first-order mutation fills each hole with the same content as P except filling one hole with a mutated content.*

Definition 3 (High-order Mutation). *Given a test program P and its mutation skeleton $\mathbb{P}$, for all the n holes (e.g., $\{\square_1, \square_2, ..., \square_n\}$) within $\mathbb{P}$, high-order mutation fills i ($2 \leq i \leq n$) holes with mutated contents, while filling the rest holes with the same content as P.*

Since we need diverse execution traces representing different ways to flip the failing test program for effective compiler bug isolation, for each mutation category we design a plurality of mutation rules. The detailed mutation rules for each mutation category are shown in Table 1. Based on the three types of holes, we design 132 specific mutation rules in total and the full list can be found in the project webpage. Here we regard each specific mutation operation on one type of program holes as an individual mutation rule, e.g, replacing "`x=0`" with "`y=0`" and replacing "`x=0`" with "`x=1`" are two different rules, while replacing "`x=0` with "`y=0`" and replacing "`x=0`" with "`z=0`" belong to the same rule.

$$\begin{aligned}
\llbracket a \rrbracket_v &::= \Box_v \mid n \mid op_u\ \llbracket a \rrbracket_v \mid \llbracket a_1 \rrbracket_v\ op_a\ \llbracket a_2 \rrbracket_v \\
\llbracket b \rrbracket_v &::= \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid \\
&\quad \llbracket b_1 \rrbracket_v\ op_l\ \llbracket b_2 \rrbracket_v \mid \llbracket a_1 \rrbracket_v\ op_r\ \llbracket a_2 \rrbracket_v \\
\llbracket S \rrbracket_v &::= \Box_v := \llbracket a \rrbracket_v \mid \llbracket S_1 \rrbracket_v;\ \llbracket S_2 \rrbracket_v \mid \\
&\quad \textbf{while}\ (\llbracket b \rrbracket_v)\ \textbf{do}\ \llbracket S \rrbracket_v \mid \\
&\quad \textbf{if}\ (\llbracket b \rrbracket_v)\ \textbf{then}\ \llbracket S_1 \rrbracket_v\ \textbf{else}\ \llbracket S_2 \rrbracket_v
\end{aligned}$$

(a) Syntax rules for variable mutations

$$\begin{aligned}
\llbracket a \rrbracket_o &::= x \mid n \mid \Box_o\ \llbracket a \rrbracket_o \mid \llbracket a_1 \rrbracket_v\ \Box_o\ \llbracket a_2 \rrbracket_o \\
\llbracket b \rrbracket_o &::= \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid \\
&\quad \llbracket b_1 \rrbracket_o\ \Box_o\ \llbracket b_2 \rrbracket_o \mid \llbracket a_1 \rrbracket_o\ \Box_o\ \llbracket a_2 \rrbracket_o \\
\llbracket S \rrbracket_o &::= x := \llbracket a \rrbracket_o \mid \llbracket S_1 \rrbracket_o;\ \llbracket S_2 \rrbracket_o \mid \\
&\quad \textbf{while}\ (\llbracket b \rrbracket_o)\ \textbf{do}\ \llbracket S \rrbracket_o \mid \\
&\quad \textbf{if}\ (\llbracket b \rrbracket_o)\ \textbf{then}\ \llbracket S_1 \rrbracket_o\ \textbf{else}\ \llbracket S_2 \rrbracket_o
\end{aligned}$$

(b) Syntax rules for operator mutations

$$\begin{aligned}
\llbracket a \rrbracket_c &::= x \mid \Box_c \mid op_u\ \llbracket a \rrbracket_c \mid \llbracket a_1 \rrbracket_c\ op_a\ \llbracket a_2 \rrbracket_c \\
\llbracket b \rrbracket_c &::= \textbf{true} \mid \textbf{false} \mid \textbf{not}\ b \mid \\
&\quad \llbracket b_1 \rrbracket_c\ op_l\ \llbracket b_2 \rrbracket_c \mid \llbracket a_1 \rrbracket_c\ op_r\ \llbracket a_2 \rrbracket_c \\
\llbracket S \rrbracket_c &::= x := \llbracket a \rrbracket_c \mid \llbracket S_1 \rrbracket_c;\ \llbracket S_2 \rrbracket_c \mid \\
&\quad \textbf{while}\ (\llbracket b \rrbracket_c)\ \textbf{do}\ \llbracket S \rrbracket_c \mid \\
&\quad \textbf{if}\ (\llbracket b \rrbracket_c)\ \textbf{then}\ \llbracket S_1 \rrbracket_c\ \textbf{else}\ \llbracket S_2 \rrbracket_c
\end{aligned}$$

(c) Syntax rules for constant mutations

Figure 3: Skeletal program structures for the WHILE-style language

Table 1: Summary of mutation rules

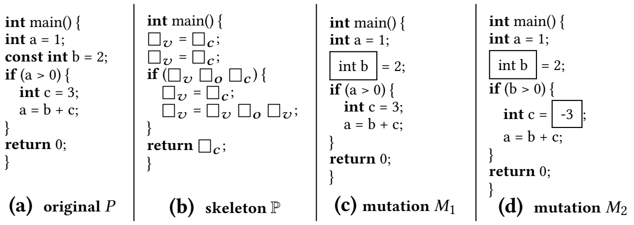
"Holes"	Mutation Rules
$\Box_v$	Inserting/removing a qualifier, i.e., `volatile`, `const`, and `restrict`; Inserting/removing/replacing a modifier, i.e., `long`, `short`, `signed`, `unsigned`; Replacing a variable by another variable within the feasible scope;
$\Box_o$	Replacing a binary operator by another binary operator within the same category, e.g., arithmetic, relational, and logical operators; Replacing/removing a unary operator, i.e., prefix increment, postfix increment, prefix decrement, postfix decrement, and logical negation;
$\Box_c$	Changing the value of an integer constant via a typical operation, i.e., `value+1`, `value-1`, `value*0`, and `value*(-1)`;

Example. We use an example to illustrate our witnessing mutation shown in Figure 4. Figure 4a shows an original program P. Figure 4b shows the mutation skeleton $\mathbb{P}$ of P. Figures 4c and 4d show two example mutated programs by filling holes in $\mathbb{P}$, where the former is a *first-order* mutation M_1 and the latter is a *high-order* mutation M_2. The mutated holes have been highlighted within boxes.

Mutation Outcomes. After generating a test program via mutation, it is essential to judge whether it is passing or not. There are two types of compiler bugs: *crash* and *wrong-code* bugs [75, 82]. The former denotes that the compiler crashes when compiling a test program under some compilation options; while the latter mainly denotes that the compiler miscompiles a program, causing it to produce inconsistent execution results without any failure messages under different compilation options. Different types of bugs require different test oracles, which determine whether a test program is passing or not. If the given program triggers a crash bug, the used oracle is whether the compiler crashes again under the same compilation options. If the given program triggers a wrong-code bug, the used oracle is whether a generated program still produces inconsistent results across prior inconsistent compilation options. DiWi does not apply mutations on the code used as oracles (e.g., `printf` statements in C), since it may cause the fake passing program problem, i.e., the constructed passing program via such mutations may not be really passing but the used oracle simply cannot reveal the bug. More discussion about it is presented in Section 6.

4.2 Heuristic-Based Test Program Generation

Due to huge search space and limited computational resources, we cannot generate all witness test programs via mutation and then select a subset of effective witness test programs from them. One of the most cost-effective ways is that, during each generation, we generate a witness test program that differs from existing ones as much as possible. In DiWi, given a failing test program, different mutation rules are not equally effective. The mutation rules that more frequently generate diverse witness test programs should be selected with higher probability for further mutations. Based on this insight, we propose our heuristic-based test program generation in the following subsections. In particular, we use *coverage distance* to measure the diversity between test programs:



```
int main() {
int a = 1;
const int b = 2;
if (a > 0) {
   int c = 3;
   a = b + c;
}
return 0;
}
```

(a) original P

```
int main() {
□_v = □_c;
□_v = □_c;
if (□_v □_o □_c) {
   □_v = □_c;
   □_v = □_v □_o □_v;
}
return □_c;
}
```

(b) skeleton $\mathbb{P}$

```
int main() {
int a = 1;
int b = 2;
if (a > 0) {
   int c = 3;
   a = b + c;
}
return 0;
}
```

(c) mutation M_1

```
int main() {
int a = 1;
int b = 2;
if (b > 0) {
   int c = -3;
   a = b + c;
}
return 0;
}
```

(d) mutation M_2

Figure 4: Example of witnessing mutation

Definition 4 (Coverage Distance). *The distance Dist between two test programs P_1 and P_2 is the Jaccard distance between their statement coverage (where $Stmt_{P_1}$ and $Stmt_{P_2}$ represent the set of covered statements by P_1 and P_2 respectively):*

$$Dist(P_1, P_2) = 1 - \frac{Stmt_{P_1} \cap Stmt_{P_2}}{Stmt_{P_1} \cup Stmt_{P_2}} \tag{1}$$

During the process of constructing a set of effective witness test programs, DiWi first selects a seed test program to mutate (described in Section 4.2.1), and then selects a mutation rule to apply (described in Section 4.2.2) in each iteration.

4.2.1 Seed Program Selection. The initial seed test program is the given failing test program. All witness test programs are derived from this initial seed test program, by conducting first-order or high-order mutations on it. Actually, n_{th}-order mutations can be regarded as conducting first-order mutations on the programs generated by $(n\text{-}1)_{th}$-order mutations on the initial seed test program. That is, DiWi also treats the generated test programs via mutation as seed test programs for following iterations. Here we can call the initial seed test program the 0_{th}-order mutation.

To reduce the risk of introducing failing test programs that are due to other bugs, DiWi first selects $(n-1)_{th}$-order (n ≥ 1) failing test programs as seed test programs. If it cannot construct any witness test program under the given terminating condition, DiWi then selects n_{th}-order (n ≥ 1) failing test programs. The reason is that higher order failing test programs are more likely to incur other bugs. Moreover, lower order failing test programs help control the trace similarity between the newly generated witness test program and the given failing test program. Please note that DiWi rejects any newly generated witness test program without increasing diversity.

4.2.2 Mutation Rule Selection. Based on a selected seed program, DiWi selects a mutation rule to mutate it. However, these mutation rules are not equally effective to generate diverse witness test programs for a given failing test program. Also, the same mutation rule performs differently for different initial failing test programs. Therefore, we carefully design an adaptive procedure to select mutation rules for constructing a set of effective witness test programs. Intuitively, if a mutation rule can more frequently generate witness test programs that have greater diversity with the existing ones, the mutation rule should be selected with higher probability for further mutations. Based on the intuition, we compute a priority score for each mutation rule MR as $Score(MR) = \left(\frac{1}{m}\sum_{i=1}^{m} Dist(P, P_i)\right) * Rate_s$, where: ① m is the number of existing witness test programs in the set; ② P is the new witness test program generated by using MR; ③ $Dist(,)$ is the coverage distance, computed by Formula 1; ④ $Rate_s$ is the success rate of generating accepted witness test programs, i.e., the ratio of the number of times the witness test program generated by MR is accepted into the witness set to the number of times MR is selected for mutations.

Mutation rules can be ranked according to the descending order of the computed priority scores. However, we cannot directly select the mutation rule ranked at the 1_{st} position for next mutation, since the ranking is based on historical results of these mutation rules and can hardly perfectly predict future results. Therefore, each mutation rule should have some probability to be selected for next mutation, and a mutation rule ranked higher in the ranking list should have a larger probability to be selected. That is, the problem of mutation rule selection in DiWi can be regarded as *the sampling problem from a probability distribution.*

Here, since which mutation rule to be selected depends on the most recent behavior of each mutation rule, it is actually a typical Markov Chain (MC). Therefore, to solve the sampling problem from a probability distribution, DiWi adopts the Metropolis-Hastings (MH) algorithm [40], the most popular Markov Chain Monte Carlo method, as heuristic by assuming the desired distribution to be equilibrium distribution [23]. Here, MH obtains random samples from a probability distribution. In our context, it samples the next mutation rule (denoted as MR_b) based on the current mutation rule (denoted as MR_a) according to a probability distribution. If MR_b is better than MR_a, i.e., the priority score of MR_b is larger than that of MR_a, MR_b is definitely accepted; If not, MR_b still has some probability to be accepted. Following the existing work [20], we set the probability distribution to be the geometric distribution, which is the probability distribution of the number X of Bernoulli trials needed to obtain one success. If the success probability on each trial is p, the probability the k_{th} trial being the first success can be computed as $Ps(X = k) = (1 - p)^{k-1}p$.

During the process, mutation rules are selected randomly, and thus the proposal distribution is symmetric. Therefore, the probability of accepting MR_b given MR_a is computed as $Pa(MR_b|MR_a) = \frac{Ps(MR_b)}{Ps(MR_a)} = (1-p)^{k_b - k_a}$, where k_a and k_b are the positions of MR_a and MR_b in the ranking list of mutation rules according to the descending order of their priority scores. Please note that, when MR_b is better than MR_a (i.e., $Ps(MR_b) > Ps(MR_a)$), $Pa(MR_b|MR_a) = 1$. After acquiring the result of MR_b (i.e., accept or reject), DiWi updates its score and re-ranks these mutation rules for next iteration.

Algorithm 1: Heuristic-based Test Program Generation

```
   Input  : $\mathcal{S}$: Seed test-program set $\{P_f\}$
            $\mathcal{MR}$: A list of mutation rules $[MR_i | i \in 1 \ldots 132]$
   Output : $\mathcal{P}$: A set of witness test programs
 1 $MR_a \leftarrow MR_{i \leftarrow random(1 \ldots 132)}$
 2 $\mathcal{S}' \leftarrow [\,]$
 3 while not termination do
 4     $P \leftarrow select(\mathcal{S})$ /* select a program from $\mathcal{S}$ */
 5     $k_a \leftarrow position(MR_a)$ /* get $MR_a$'s position in $\mathcal{MR}$ */
 6     do
 7         $MR_b \leftarrow MR_{i \leftarrow random(1 \ldots 132)}$
 8         $k_b \leftarrow position(MR_b)$
 9         $f \leftarrow random(0, 1)$
10     while $f \geq (1-p)^{k_b - k_a}$;
11     $P' \leftarrow mutate(P, MR_b)$
12     if $P'$ is passing $\wedge \forall P_i \in \mathcal{P}, D(P', P_i) \neq 0$ then
13         $\mathcal{P} \leftarrow \mathcal{P} \cup \{P'\}$
14     end
15     if $P'$ is failing then
16         $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{P'\}$
17     end
18     $updateScore(MR_b, P', \mathcal{P})$
19     $\mathcal{MR} \leftarrow sort(\mathcal{MR})$
20     $MR_a \leftarrow MR_b$
21 end
22 if $Size(\mathcal{P}) > 0$ then
23     return $\mathcal{P}$
24 end
25 else
26     $\mathcal{S} \leftarrow \mathcal{S}'$
27     Repeat from Line 2
28 end
```

4.2.3 Overall Algorithm. We formally present the generation process of DiWi in Algorithm 1. The initial set of seed programs contains only the given failing test program P_f, and the priority score of each mutation rule is 0. In this algorithm, Line 1 randomly selects a mutation rule as the current one MR_a. Lines 3-21 construct a set of witness test programs until achieving a terminating condition. Line 4 selects a seed program P to mutate. Line 5 gets the position of MR_a in the ranking list of mutation rules. Lines 6-10 acquire the next mutation rule MR_b. Line 11 applies MR_b to mutate P to generate a new program P'. Lines 12-17 determine whether P' is accepted by $\mathcal{P}$ and $\mathcal{S}'$ based on its execution results and coverage distances with the existing witness programs in $\mathcal{P}$. Lines 18-20 update the score of MR_b and re-rank these mutation rules for next iteration. If P' is accepted into $\mathcal{P}$, the score is updated by changing the two items in the formula *Score*; Otherwise, the score is updated by just changing the latter item in the formula *Score*, since the witness set and the coverage distances are not changed. Lines 22-28 determine whether terminating the construction process. If there is no witness program constructed via lower order mutation, the construction process (Lines 2-21) is repeated by using higher order mutation (i.e., using higher order failing programs as seed programs).

4.3 Aggregation-Based Compiler Bug Isolation

After constructing a set of witness test programs, DiWi isolates compiler bugs by analyzing the set of witness test programs and the given failing test program. Following SBFL [7], DiWi computes the suspicious value for each statement within the touched code when compiling the given failing test program. Here DiWi adopts Ochiai [7], one of the most effective formulae in SBFL, to compute the suspicious value for each touched statement. The formula is $sus(s) = \frac{ef_s}{\sqrt{(ef_s + nf_s)(ef_s + ep_s)}}$, where ef_s and nf_s represent the number of failing test programs that execute and do not execute statement s, and ep_s represent the number of passing test programs executing s. Here, we have only one given failing test program and just consider the statements touched by the failing test program, and thus ef_s is 1 and nf_s is 0. Therefore, $sus(s) = \frac{1}{\sqrt{1+ep_s}}$ in DiWi.

Then, similar to method-level aggregation [74], DiWi computes the suspicious value of each file by aggregating the suspicious values of all the touched statements in the file. When a failing test program is mutated to a set of witness test programs, the coverage for the buggy file would be changed more than that for the bug-free files on the whole. Therefore, to compute the suspicious value of each file, we use the formula $SUS(f) = \frac{\sum_{i=1}^{n_f} sus(s_i)}{n_f}$, where n_f is the number of touched statements when compiling the failing test program in the file f.

5 EVALUATION

In this study, we address the following research questions:

- **RQ1**: How does DiWi perform on compiler bug isolation?
- **RQ2**: Do our search-based witness test programs outperform the developer-provided test suite and the randomly generated passing test programs?
- **RQ3**: Does our heuristic-based search strategy outperform the random search strategy during mutation?
- **RQ4**: Can DiWi take advantage of the developer-provided test suite (which is an exploration to further boost DiWi)?

5.1 Benchmark

We used GCC and LLVM as subjects, which cover almost all popular C compilers used in the existing work [15, 17, 43, 44, 82]. To investigate the effectiveness of DiWi, from the bug repositories [3, 6] of GCC and LLVM, we manually collected 90 bugs in total, each of which is required to satisfy the following conditions: 1) the bug has the equipped failing test program and the compilation options triggering it in the bug report; 2) the bug has been fixed; 3) the bug can be reproduced in our experimental environment. We manually collected bugs reported after 2013 following these conditions until we had 45 bugs for each compiler since the manual process is costly. Also, we manually identified the buggy locations (i.e., buggy files) for each bug, which serve as the ground truth to evaluate the effectiveness of bug isolation in our study. On average, a GCC buggy version has 1,588 files with 1,414K source lines of code (SLOC), while a LLVM buggy version has 3,507 files with 1,431K SLOC[1]. We release the benchmark to facilitate the future research on compiler bug detection, isolation, and fixing, and welcome more researchers to contribute to this benchmark. In our benchmark, each bug has: ① *buggy compiler version*; ② *failing test program*; ③ *compilation options for reproducing the bug*; ④ *buggy location*; ⑤ *fixed version*. Our benchmark and code are available at the project webpage: https://github.com/JunjieChen/DiWi.

5.2 Implementation and Configuration

DiWi utilizes Clang Libtooling library [1] to parse a test program to an abstract syntax tree (AST) and then mutates it at the AST. DiWi utilizes Gcov [4] to collect compiler test coverage. Here we set the success probability of each Bernoulli trial p in Algorithm 1 to be 0.023 by satisfying the following conditions: ① $0.95 \leq \sum_{k=1}^{132}(P(X = k)) \leq 1$; ② $p \geq \frac{1}{132}$; ③ $(1-p)^{132-1}p < \varepsilon$, where ε is a quite small deviation (e.g., 0.001). We set the terminating condition of DiWi to be one hour limit in our study. For any technique involving randomness, we repeated it 5 times and use the median results. Our study is conducted on a workstation with four-core CPU, 120G memory, and Ubuntu 14.04 operating system.

5.3 Measurements

To evaluate the effectiveness of bug isolation, we measure the position of each buggy file in the ranking list produced by a bug isolation technique. If more than two files have the same suspiciousness, we use the worst ranking following the existing work [37, 51, 68]. We compute the following widely used metrics [9, 51, 60, 74]:

Top-n refers to the number of successfully isolated bugs within the Top-n position (i.e., $n \in \{1, 5, 10, 20\}$ in our study) in the ranking list. Larger is better.

Mean First Rank (MFR) refers to the mean of the first buggy file rank for each bug. This metric emphasizes fast isolation of the first buggy element to ease debugging. Smaller is better.

Mean Average Rank (MAR) refers to the mean of the average rank of all buggy files for each bug. Different from MFR, MAR emphasizes precise isolation for all buggy elements.

5.4 Compared Techniques

Spectrum-based bug isolation (SBFL) [80] is the most widely-studied bug isolation technique among traditional bug isolation techniques. It is interesting to evaluate the effectiveness of traditional SBFL on compilers. It first records the coverage status of each program element (i.e., each compiler file in the study) during each test execution and the test outcomes (i.e., passing or failing). Then SBFL computes a suspicious value for each program element using some formula, and finally ranks the program elements based on their suspicious values. Researchers have made dedicated efforts to design various formulae on suspiciousness computation. We evaluated eight popular formulae following the existing work [68, 81, 88], including *SBI*, *Ochiai*, *Tarantula*, *Jaccard*, *Ochiai2*, *Kulczynski2*, O^P, and D^2, and found they achieved extremely similar results for compiler bug isolation in our study. Due to space limitation, we use the most effective *Ochiai* as the representative. For each compiler bug, we use the given failing test program as the failing test program, and use the developer-provided test suite as the passing test programs.

As the core of DiWi is to generate effective witness test programs, it is interesting to *investigate the impact of generated witness test programs on bug isolation.* To achieve this goal, we replace the set of our search-based witness test programs with the developer-provided test suite, and then use the aggregation-based SBFL to isolate bugs. We call this technique $\textbf{SBFL}_a^{dev}$. Besides, randomly generated test programs via test-program generation tools like Csmith [82] have been demonstrated to be quite effective for detecting compiler bugs [15, 82]. Therefore, we also replace the set of our search-based witness test programs with a set of randomly generated passing test programs via Csmith [82], and then use the aggregation-based SBFL to isolate bugs. Similarly, we call this technique $\textbf{SBFL}_a^{rand}$.

Besides, to *investigate the impact of our heuristic-based search strategy on bug isolation*, we replace this strategy with the random search strategy. That is, we do not have any guidance for constructing witness test programs via mutation. We call this variant of DiWi $\textbf{DiWi}^{rand}$ (random search). Note that we used the same terminating condition for all compared techniques for fair comparison.

[1] Since GCC and LLVM are implemented using C and C++ respectively, we consider all the C files for the former and C++ files for the latter.

5.5 Threats to Validity

First, the findings in this work may not generalize to other compiler bugs. To reduce this threat, we tried our best to construct a new dataset including 90 real-world compiler bugs. Note that the process was extremely time consuming and tedious; to our knowledge this is the largest dataset for reproducible compiler bugs. Second, besides the used two kinds of passing test programs, there are other kinds of test programs, e.g., programs generated via swarm testing [29]. In the future, we will use more kinds of passing programs for comparison. Third, the settings (e.g., parameters in our search-based strategy and terminating condition of DiWi) may impact our study. In the future, we will explore their impacts.

5.6 Results and Analysis

5.6.1 Overall effectiveness of DiWi. Rows "DiWi" in Table 2 present the effectiveness of DiWi. Overall, DiWi successfully isolates 10/37/60/71 bugs (out of 90 bugs) within Top-1/5/10/20 buggy files, demonstrating its effectiveness on compiler bug isolation. That is, about 66.67% and 78.89% bugs are effectively isolated within 10 and 20 compiler files, respectively. We further analyzed the effectiveness of DiWi for different compiler systems. Intuitively, LLVM has a much larger number of files (shown in Section 5.1), and should be harder to perform bug isolation. However, interestingly, shown in Table 2, DiWi achieves quite similar effectiveness on GCC and LLVM, and DiWi even isolates 3 more bugs within Top-5 on LLVM than GCC. Moreover, we find that other studied techniques indeed perform worse on LLVM than GCC. Therefore, that demonstrates the scalability of DiWi — DiWi's effectiveness does not decrease dramatically when facing larger compiler systems.

We also analyzed the comparison effectiveness between DiWi and traditional SBFL shown in Rows "SBFL" in Table 2. We find that traditional SBFL performs poorly on the studied compilers. For example, SBFL ranks the GCC buggy files as the 276.22_{th} position, while ranking the LLVM buggy files as the 619.09_{th} position on average. To our knowledge, this is the first study demonstrating that the intensively studied SBFL cannot scale to real-world compiler systems. On the contrary, although equally simple and lightweight, our DiWi is able to significantly outperform SBFL. On average, DiWi localizes compiler buggy files within 14.27(MFR) and 14.76(MAR), outperforming SBFL by 96.81%(MFR) and 96.70%(MAR).

Qualitative Analysis. We further conducted qualitative analysis using two examples. Figure 5 shows an example LLVM bug, where the left is the failing test program and the right is one of witness test programs generated by DiWi. The test program (in Figure 5a) is miscompiled by the LLVM trunk (revision 229830) at `-O1` and above. The bug occurs at the file "`ScalarEvolution.cpp`", which incorrectly promotes the 16-bit *add* into a 32-bit *add*. After just one mutation shown in Figure 5b, the mutated program does not trigger the bug, demonstrating the power of our designed mutation rules. We computed the coverage distance between the two programs, and its value is only 0.013, which confirms our assumption that minor changes are likely to make them share close compiler execution traces. In particular, DiWi ranks the buggy file at the $\mathbf{6}_{th}$ of all files.

```
unsigned short a = 1;
int b = 65536;
int c;
int main () {
  for (c = 0; c < 1; c = 1) {
    for (;;) {
      b &= --a;
      break;
    }
  }
  if (b)
    __builtin_abort ();
  return 0;
}
```

(a) Failing test program

```
unsigned short a = 1;
int b = 65536;
int c;
int main () {
  for (c = 0; c < 1; c = 1) {
    for (;;) {
      b &= a++;
      break;
    }
  }
  if (b)
    __builtin_abort ();
  return 0;
}
```

(b) Passing mutation

Figure 5: LLVM Bug 22641

```
int printf (const char *, ...);
int a[6], b, c = 1, d;
short e;
void fn1 (int p) {
  b = a[p];
}
int main () {
  a[0] = 1;
  if (c)
    e--;
  d = e;
  long long f = e;
  fn1 ((f >> 56) & 1);
  printf ("%d\n", b);
  return 0;
}
```

(a) Failing test program

```
int printf (const char *, ...);
int a[6], b, c = 1, d;
volatile short e;
void fn1 (int p) {
  b = a[p];
}
int main () {
  a[0] = 1;
  if (c)
    e--;
  d = e;
  long long f = e;
  fn1 ((f >> 56) & 1);
  printf ("%d\n", b);
  return 0;
}
```

(b) Passing mutation

Figure 6: GCC Bug 59747

Figure 6 shows an example for a GCC bug. The failing test program in Figure 6a is miscompiled by the GCC trunk (revision 206472) at `-Os` and above. The bug occurs at the file "`tree-ssa-sink.c`", because it does not well handle the case where a second eliminated extension requires widening a copy created for elimination of a prior extension. After adding "`volatile`" as shown in Figure 6b, the mutated program does not trigger the bug anymore. We computed their coverage distance, and its value is 0.019, which is also very small. In particular, DiWi ranks the buggy file at the $\mathbf{1}_{st}$ of all files.

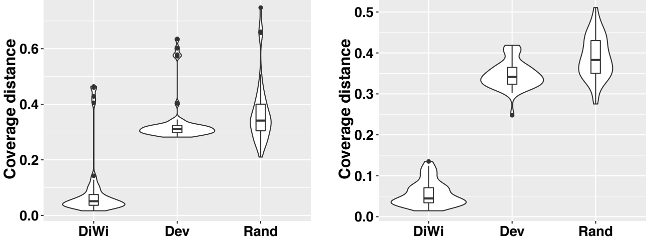
Furthermore, DiWi also brings extra benefits. That is, these generated witness test programs via mutation provide some useful hints for the developers to facilitate bug diagnosis/fixing. For example, from Figure 5, the bug is not triggered again by replacing "- -" with "++", which means that "-1" or "+1" has an impact on the bug. That is true, because incorrect promotion from 16-bit *add* to 32-bit *add* leads to incorrect widening from "-1" to "65535".

5.6.2 DiWi v.s. SBFL$_a^{dev}$ v.s. SBFL$_a^{rand}$. We compared DiWi with SBFL$_a^{dev}$ and SBFL$_a^{rand}$ to *investigate the impact of our search-based witness test programs.* From Rows "DiWi", "SBFL$_a^{dev}$", and "SBFL$_a^{rand}$" in Table 2, DiWi outperforms them in terms of all the used metrics. For example, DiWi isolates 150.00%/68.18%/33.33%/12.70% more bugs than SBFL$_a^{dev}$ and 66.67%/54.17%/36.36%/18.33% more bugs than SBFL$_a^{rand}$, within Top-1/5/10/20 for all the studied bugs. That demonstrates that DiWi achieves much more precise isolation results than SBFL$_a^{dev}$ and SBFL$_a^{rand}$. In particular, we conducted the Wilcoxon Signed-Rank Test [79] for their isolation ranks at the significance level of 0.05 to determine whether there are significant differences between them on all the studied bugs. The p-value is 0.0016 and 0.0021 respectively, demonstrating that DiWi does significantly outperform them. These results indicate that our search-based witness test programs are more powerful than both the developer-provided and randomly generated (via Csmith) witness test programs on isolating compiler bugs.

Table 2: Compiler bug isolation effectiveness comparison

Subject	Technique	Top-1	$\Uparrow_{Top-1}$	Top-5	$\Uparrow_{Top-5}$	Top-10	$\Uparrow_{Top-10}$	Top-20	$\Uparrow_{Top-20}$	MFR	$\Uparrow_{MFR}$	MAR	$\Uparrow_{MAR}$
GCC	DiWi	5	–	17	–	30	–	36	–	13.93	–	14.76	–
	SBFL	0	∞	0	∞	0	∞	0	∞	276.22	94.96	276.22	94.66
	SBFL_a^{dev}	3	66.67	12	41.67	25	20.00	30	20.00	17.29	19.43	18.07	18.32
	SBFL_a^{rand}	1	400.00	12	41.67	22	36.36	29	24.14	17.66	21.12	19.62	24.77
	DiWi^{rand}	3	66.67	14	21.43	24	25.00	33	9.09	16.91	17.62	17.57	16.51
LLVM	DiWi	5	–	20	–	30	–	35	–	14.60	–	14.76	–
	SBFL	1	400.00	3	566.67	3	900.00	3	1,066.67	619.09	97.64	619.09	97.62
	SBFL_a^{dev}	1	400.00	10	100.00	20	50.00	33	6.06	26.44	44.78	26.61	44.53
	SBFL_a^{rand}	5	–	12	66.67	22	36.36	31	12.90	24.02	39.22	24.21	39.03
	DiWi^{rand}	3	66.67	15	33.33	24	25.00	33	6.06	20.82	29.88	21.04	29.85
ALL	DiWi	10	–	37	–	60	–	71	–	14.27	–	14.76	–
	SBFL	1	900.00	3	1,133.33	3	1,900.00	3	2,266.67	447.66	96.81	447.66	96.70
	SBFL_a^{dev}	4	150.00	22	68.18	45	33.33	63	12.70	21.87	34.75	22.34	33.93
	SBFL_a^{rand}	6	66.67	24	54.17	44	36.36	60	18.33	20.84	31.53	21.92	32.66
	DiWi^{rand}	6	66.67	29	27.59	48	25.00	66	7.58	18.87	24.38	19.31	23.56

[*] Columns "$\Uparrow_*$" present the improvement rates of **DiWi** over a compared technique in terms of various measurements.



(a) GCC **(b) LLVM**

Figure 7: Coverage distance between generated witness test programs and the given failing test program

To further investigate why our search-based witness test programs can significantly outperform the others, we further quantitatively evaluated one basic assumption of DiWi, i.e., minor mutation changes are likely to make them share close compiler traces. We computed the mean coverage distance between the given failing test program and all witness test programs for each bug. Figure 7 shows the coverage distance comparison among DiWi (our search-based witness program generation), SBFL_a^{dev} (the developer-provided test suite, denoted as *Dev* in this figure), and SBFL_a^{rand} (the Csmith random program generation, denoted as *Rand*). In this figure, the violin plots show the density of coverage distances at different values, and the box plots show the median and interquartile ranges. From this figure, we find that our search-based witness programs have much smaller coverage distances with the given failing test program than the developer-provided witness programs and the randomly generated witness programs via Csmith. That validates our assumption. Interestingly, in GCC there are some cases where the coverage distances are obviously larger than other cases. We looked into the code and found that all these bugs are *crash* bugs, and various code regions not executed by the failing test program (due to crashes) can be executed by the witness test programs.

5.6.3 DiWi vs. $DiWi^{rand}$. We also compared DiWi and DiWi^{rand} to *investigate the impact of our heuristic-based search strategy* shown in Rows "DiWi^{rand}" in Table 2. From this table, DiWi outperforms DiWi^{rand} for both GCC and LLVM in terms of all the used metrics. DiWi isolates 66.67%/27.59%/25.00%/7.58% more bugs within Top-1/5/10/20 than DiWi^{rand} for all the studied bugs. DiWi performs 24.38% and 23.56% better than DiWi^{rand} in terms of MFR and MAR. That demonstrates that DiWi performs more precise than DiWi^{rand} for isolating both GCC and LLVM bugs. We also conducted the Wilcoxon Signed-Rank Test for the isolation ranks of DiWi and DiWi^{rand}. The p-value is 1.345e-06, demonstrating the superiority of DiWi over DiWi^{rand}. The results indicate that our heuristic-based search strategy outperforms random search.

To investigate why the heuristic-based search strategy outperforms the random search strategy, we quantitatively evaluated another basic assumption of DiWi, i.e., witness test programs generated via our heuristic-based search strategy should have great diversity. We computed the coverage diversity among witness test programs. Here we first computed the minimum distance for each witness test program with others, and then computed the mean of all minimum distances for all witness test programs. We find that the mean coverage diversity of our heuristic-based strategy is about 214.72X and 14.10X greater than that of the random search strategy for GCC and LLVM, respectively. That is, DiWi indeed has greater diversity than DiWi^{rand}, validating our assumption.

5.6.4 Exploring DiWi with Developer Tests. Our search-based witness test programs have been demonstrated to outperform the other two. We further analyzed the cases where each type of witness programs performs well. Here we chose the bugs isolated within Top-5 to analyze. We found although DiWi isolates 37 bugs within Top-5, SBFL_a^{dev} and SBFL_a^{rand} can also isolate 11 additional bugs in total within Top-5. That is, the developer-provided programs and randomly generated programs via Csmith can *complement* our search-based witness programs to some degree. If we can effectively synthesize them, the compiler bug isolation effectiveness may be improved. We analyzed why randomly generated programs can isolate additional bugs, and found the reason to be that the tool Csmith cannot cover all C language features, causing some files always uncovered. When the bug occurs at these files, due to such an *occasional* factor, the files are easy to isolate using Csmith generated passing programs. Getting rid of this occasional factor, we made the first attempt in the direction by synthesizing our search-based witness programs and the developer-provided witness programs.

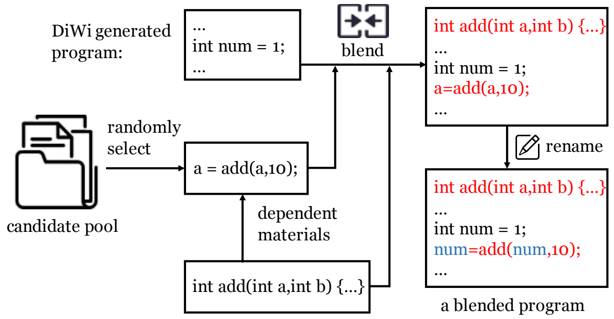


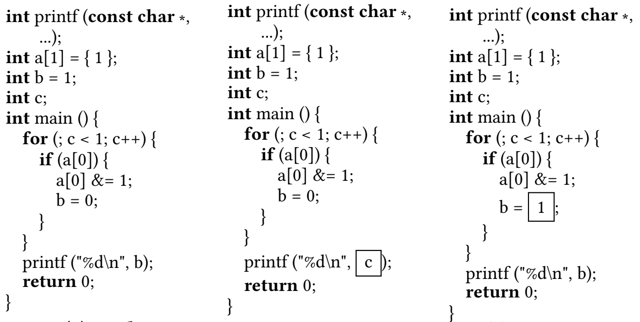
Figure 8: Test program synthesis

For each search-based witness test program by DiWi, the synthesis technique produces a blended witness program automatically as follows: (1) it treats all basic blocks/statements in all developer-provided programs as a candidate pool; (2) it randomly selects a candidate block/statement from the pool and collects its all dependent materials (e.g., method declaration and header file); (3) it randomly inserts the candidate block/statement and its dependent materials to the search-based witness program, and conducts refactoring for new variables in the block/statement to make the blended program valid, i.e., renaming them to the variables occurred in the original program with compatible types. Figure 8 shows the process of generating a blended program. We used Clang Libtooling library [1] to conduct such synthesis at the AST, and the whole process is fully automated. We repeated the above steps for each search-based witness program until a blended witness program is produced. In this way, we get a set of blended witness programs. We then used the aggregation mechanism to isolate compiler bugs based on the given failing program and the blended witness programs.

We evaluated whether the synthesis can further improve DiWi. From the results, among 90 compiler bugs, the synthesis improves the isolation effectiveness for 50% bugs (45 out of 90), and reduce the effectiveness for only 10 bugs. Also, such synthesis effectively isolates 30.00% and 24.32% more bugs within Top-1 and Top-5 files than DiWi, respectively. Its improvement rates of MFR and MAR are 11.75% and 9.15%, respectively, compared with DiWi. This is because the synthesis provides more possibilities for DiWi to find effective witness test programs by augmenting the mutation space, demonstrating a promising future to further explore effective ways for blending test programs from different sources.

6 DISCUSSION

What developers want. To investigate the practicability of DiWi, we conducted a survey by communicating with 7 compiler developers (sending out 10 requests in total) from 4 international companies building their own compilers (including the LLVM team). 6 developers confirmed that their compiler bug debugging process starts from buggy files identification and this step is time-consuming, indicating the necessity of compiler bug isolation at the file level. Moreover, 6 developers think the effectiveness of DiWi (shown in our study) is practical and show strong desire for DiWi by using the words "can't wait to see" during the communications. Even the developer who does not first identify buggy files when debugging, also expresses his/her willing to improve the debugging process by using DiWi. In the future, we will improve DiWi at finer granularity such as the method level.



```
int printf (const char *,
     ...);
int a[1] = { 1 };
int b = 1;
int c;
int main () {
  for (; c < 1; c++) {
    if (a[0]) {
      a[0] &= 1;
      b = 0;
    }
  }
  printf ("%d\n", b);
  return 0;
}
```

(a) Failing

```
int printf (const char *,
     ...);
int a[1] = { 1 };
int b = 1;
int c;
int main () {
  for (; c < 1; c++) {
    if (a[0]) {
      a[0] &= 1;
      b = 0;
    }
  }
  printf ("%d\n", c );
  return 0;
}
```

(b) Fake pass-1

```
int printf (const char *,
     ...);
int a[1] = { 1 };
int b = 1;
int c;
int main () {
  for (; c < 1; c++) {
    if (a[0]) {
      a[0] &= 1;
      b = 1 ;
    }
  }
  printf ("%d\n", b);
  return 0;
}
```

(c) Fake pass-2

Figure 9: Example of test oracle challenge (GCC Bug 61140)

Test oracle challenge. We present the used test oracles to determine whether a mutated program is passing in Section 4.1. However, such oracles are not absolutely precise, especially for wrong-code bugs. DiWi treats the mutated program producing consistent results as a passing program, but it may still trigger the bug. For example, Figure 9a shows a failing program, where the outputs (of b) under -O0 and -O1 are different. The other two figures are two mutated passing programs under the used oracle. However, both of them are *fake* passing programs. In Figure 9b, the variable b in printf statement is mutated to be c. It still triggers the bug, but it is regarded as a passing program since c prints the same results under -O0 and -O1. In Figure 9c, the value of b (i.e., 0) is mutated to be 1, which is equal to the initial value of b. Such mutations make the output always the same (i.e., 1), causing the used oracle to miss the bug. Fake passing programs may impact the isolation effectiveness. However, it is challenging to solve this oracle problem. To reduce its impact, DiWi avoids the mutations that directly change the oracle (i.e., print statements), but cannot deal well with other cases such as Figure 9c. In the future, we will introduce advanced data- and control-flow analysis [63] to address this challenge.

Undefined behavior challenge. Another challenge lies in undefined behaviors for compilers, which mean the semantics of certain operations are undefined in the programming-languages standards [30]. If a program contains undefined behaviors, compilers may produce varied results. Identifying undefined behaviors is a difficult challenge in compiler research [49, 78]. It is also a threat in our work, since our mutation may introduce undefined behaviors. However, undefined behaviors tend to impact bug detection, since different results of a "failing" program may be caused by real bugs or undefined behaviors. In DiWi, we only kept the passing programs with the same results to isolate bugs, and thus the threat may be not serious. We will relieve this problem by adopting existing light-weight methods [49] for identifying undefined behaviors.

7 CONCLUSION

In this paper, we propose a novel compiler bug isolation technique, DiWi, which proposes a heuristic-based search strategy to carefully generate a set of effective witness programs by performing our designed witnessing mutation rules on the given failing program. The results on 90 real bugs for GCC and LLVM show, DiWi isolates 78.89% of the studied bugs within 20 compiler files, significantly outperforming state-of-the-art SBFL. DiWi is general and not limited to compilers, we plan to apply it to other systems taking structurally complex test inputs, e.g., operating systems and browsers.

REFERENCES

[1] Accessed: 2019. Clang Libtooling library. http://clang.llvm.org/docs/LibTooling.html.

[2] Accessed: 2019. GCC. https://gcc.gnu.org.

[3] Accessed: 2019. GCC bug repository. https://gcc.gnu.org/bugzilla/.

[4] Accessed: 2019. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[5] Accessed: 2019. LLVM. https://llvm.org.

[6] Accessed: 2019. LLVM bug repository. https://bugs.llvm.org.

[7] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION 2007*. 89–98.

[8] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed test generation for effective fault localization. In *ISSTA*. 49–60.

[9] Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *ISSTA*. 177–188.

[10] Benoit Baudry, Franck Fleurey, and Yves Le Traon. 2006. Improving test suites for efficient fault localization. In *ICSE*. 82–91.

[11] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. 2013. Entropy-based test generation for improved fault localization. In *ASE*. 257–267.

[12] Jacqueline M. Caron and Peter A. Darnell. 1990. Bugfind: A Tool for Debugging Optimizing Compilers. *SIGPLAN Notices* 25, 1 (1990), 17–22.

[13] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. 2005. Type-based verification of assembly language for compiler debugging. In *TLDI*. 91–102.

[14] Junjie Chen. 2018. Learning to accelerate compiler testing. In *ICSE: Companion Proceeedings*. 472–475.

[15] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *ICSE*. 700–711.

[16] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test case prioritization for compilers: A text-vector based approach. In *ICST*. 266–277.

[17] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *ICSE*. 180–190.

[18] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and XIE Bing. 2018. Coverage Prediction for Accelerating Compiler Testing. *TSE* (2018). to appear.

[19] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *PLDI*, Vol. 48. 197–208.

[20] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed Differential Testing of JVM Implementations. In *PLDI*. 85–99.

[21] Vidroha Debroy and W Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *ICST*. 65–74.

[22] Nicholas DiGiuseppe and James A Jones. 2011. On the influence of multiple faults on coverage-based fault localization. In *ISSTA*. 210–220.

[23] Yadolah Dodge. 2006. *The Oxford dictionary of statistical terms.* Oxford University Press.

[24] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated Testing of Graphics Shader Compilers. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 93:1–93:29.

[25] Robert Feldt, Simon Poulding, David Clark, and Shin Yoo. 2016. Test set diameter: Quantifying the diversity of sets of test cases. In *ICST*. 223–233.

[26] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. 2008. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *ICSTW*. 178–186.

[27] Zachary P Fry and Westley Weimer. 2010. A human study of fault localization accuracy. In *ICSM*. 1–10.

[28] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *ISSTA*. to appear.

[29] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. 2012. Swarm testing. In *ISSTA*. 78–88.

[30] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. 2015. Defining the Undefinedness of C. In *PLDI*. 336–345.

[31] K. Scott Hemmert, Justin L. Tripp, Brad L. Hutchings, and Preston A. Jackson. 2003. Source Level Debugger for the Sea Cucumber Synthesizing Compiler. In *FCCM*. 228.

[32] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically Reducing Tree-structured Test Inputs. In *ASE*. 861–871.

[33] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *USENIX Security*. 445–458.

[34] Josie Holmes and Alex Groce. 2018. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *ISSRE*. 166–177.

[35] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-based fault localization for real-world multilingual programs. In *ASE*. 464–475.

[36] Reyhaneh Jabbarvand and Sam Malek. 2017. μDroid: an energy-aware mutation testing framework for Android. In *FSE*. 208–219.

[37] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault Localization Using Value Replacement. In *ISSTA*. 167–178.

[38] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*. 273–282.

[39] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ISSTA*. 433–436.

[40] Robert E. Kass, Bradley P. Carlin, Andrew Gelman, and Radford M. Neal. 1998. Markov Chain Monte Carlo in Practice: A Roundtable Discussion. *American Statistician* 52, 2 (1998), 93–100.

[41] Nico Krebs and Lothar Schmitz. 2014. Jaccie: A Java-based compiler–compiler for generating, visualizing and debugging compiler components. *Science of Computer Programming* 79 (2014), 101–115.

[42] Stephen Kyle, Hugh Leather, Björn Franke, Dave Butcher, and Stuart Monteith. 2015. Application of Domain-aware Binary Fuzzing to Aid Android Virtual Machine Testing. In *VEE*. 121–132.

[43] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *PLDI*. 216–226.

[44] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*. 386–399.

[45] Wei Le and Mary Lou Soffa. 2010. Path-based fault correlations. In *FSE*. 307–316.

[46] Wei Le and Mary Lou Soffa. 2011. Generating analyses for detecting faults in path segments. In *ISSTA*. 320–330.

[47] Wei Le and Mary Lou Soffa. 2013. Marple: Detecting faults in path segments using automatically generated analyses. *TOSEM* 22, 3 (2013), 18.

[48] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*. 3–13.

[49] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. 2017. Taming undefined behavior in LLVM. In *PLDI*. 633–647.

[50] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *ISSTA*. to appear.

[51] Xia Li and Lingming Zhang. 2017. Transforming Programs and Tests in Tandem for Fault Localization. In *OOPSLA*. 92:1–92:30.

[52] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. In *PLDI*. 15–26.

[53] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *PLDI*. 65–76.

[54] Bing Liu, Shiva Nejati, Lionel C Briand, et al. 2017. Improving fault localization for Simulink models using search-based testing and prediction models. In *SANER*. 359–370.

[55] Wes Masri. 2015. Automated Fault Localization: Advances and Challenges. In *Advances in Computers*. Vol. 99. 103–156.

[56] Wes Masri and Rawad Abou Assi. 2010. Cleansing test suites from coincidental correctness to enhance fault-localization. In *ICST*. 165–174.

[57] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[58] Hong Mei and Lu Zhang. 2018. Can big data bring a breakthrough for software automation? *SCIENCE CHINA Information Sciences* 61, 5 (2018), 056101:1–056101:3.

[59] Martin Monperrus. 2018. Automatic software repair: a bibliography. *CSUR* 51, 1 (2018), 17:1–17:24.

[60] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*. 153–162.

[61] Francisco Gomes de Oliveira Neto, Robert Feldt, Linda Erlenhov, and José Benardi de Souza Nunes. 2018. Visualizing test diversity to support test optimisation. *arXiv preprint arXiv:1807.05593* (2018).

[62] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *ICSE*. 772–781.

[63] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. Principles of program analysis. *Springer Verlag Berlin* (1999).

[64] Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. 2006. Replay compilation: improving debuggability of a just-in-time compiler. In *OOPSLA*. 241–252.

[65] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *EMSE* 14, 3 (2009), 286–315.

[66] Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate" unknown" faults. In *ICST*. 691–700.

[67] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-based Fault Localization. *STVR* 25, 5-7 (2015), 605–628.

[68] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *ICSE*. 609–620.

[69] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *PLDI*, Vol. 47. 335–346.

[70] Manos Renieres and Steven P Reiss. 2003. Fault localization with nearest neighbor queries. In *ASE*. 30–39.

[71] Jeremias Röβler, Gordon Fraser, Andreas Zeller, and Alessandro Orso. 2012. Isolating failure causes through test case generation. In *ISSTA*. 309–319.

[72] Raul Santelices, James A Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *ICSE*. 56–66.

[73] Anthony M. Sloane. 1999. Debugging Eli-Generated Compilers With Noosa. In *CC*. 17–31.

[74] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: using code and change metrics to improve fault localization. In *ISSTA*. 273–283.

[75] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *ISSTA*. 294–305.

[76] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *ICSE*. 361–371.

[77] Shaowei Wang, David Lo, Lingxiao Jiang, Hoong Chuin Lau, et al. 2011. Search-based fault localization. In *ASE*. 556–559.

[78] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*. 260–275.

[79] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1970. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Selected tables in mathematical statistics* 1 (1970), 171–259.

[80] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *TSE* 42, 8 (2016), 707–740.

[81] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *FSE*. 52–63.

[82] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. 283–294.

[83] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *FSE*. 1–10.

[84] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *TSE* 28, 2 (2002), 183–200.

[85] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *ICSM*. 23–32.

[86] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. 2010. Test generation via dynamic symbolic execution for mutation testing. In *ICSM*. 1–10.

[87] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*. 765–784.

[88] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting Spectrum-based Fault Localization Using PageRank. In *ISSTA*. 261–272.

[89] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *PLDI*. 347–361.